INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

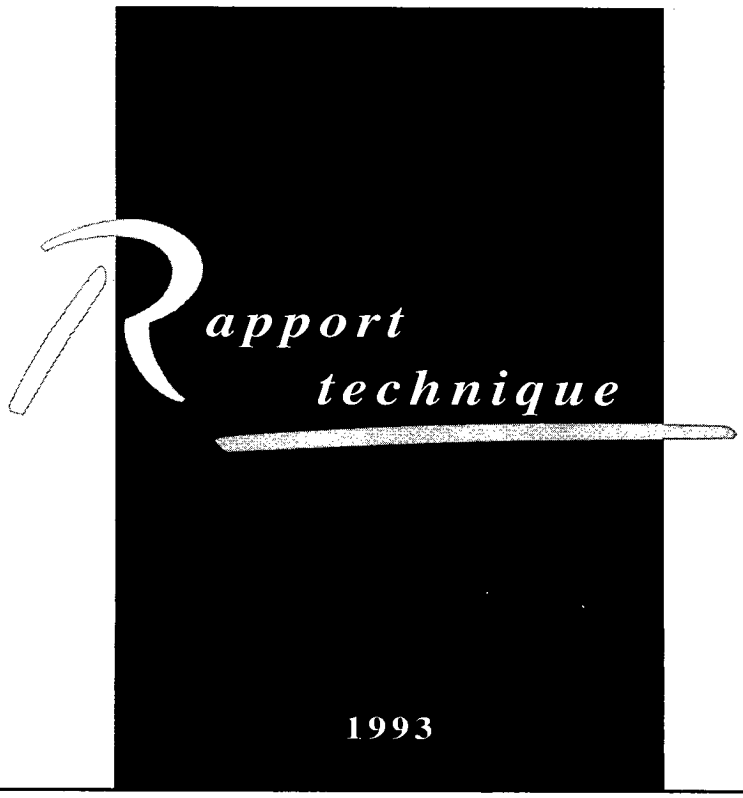
Dictionary-Matching on Unbounded Alphabets: Uniform Length Dictionaries

Dany BRESLAUER

N° 159
Septembre 1993

PROGRAMME 2

Calcul symbolique,
programmation
et génie logiciel



Dictionary-Matching on Unbounded Alphabets: Uniform Length Dictionaries

Dany Breslauer*
Institut National de Recherche en Informatique
et en Automatique
B.P. 105, 78153 Le Chesnay Cedex, France

Abstract

In the string-matching problem one is interested in all occurrences of a short pattern string in a longer text string. Dictionary-matching is a generalization of this problem where one is looking simultaneously for all occurrences of several patterns in a single text.

This paper presents an efficient on-line dictionary-matching algorithm for the case where the patterns have uniform length and the input alphabet is unbounded. A tight lower bound establishes that our approach is optimal if the only access the algorithm has to the input strings is by pairwise symbol comparisons.

In an immediate application, the new dictionary-matching algorithm can be used in a previously known higher-dimensional array-matching algorithm, improving the performance of this algorithm on unbounded alphabets. The resulting algorithm is currently the fastest known algorithm for k-dimensional array-matching on unbounded alphabets, for $k \geq 3$.

La Recherche-dictionnaire sur les alphabets non bornés: Les dictionnaires de longueurs identiques

Résumé

La recherche de motifs consiste à rechercher (toutes) les occurrences d'un motif court dans un texte plus long. La recherche-dictionnaire est une généralisation de ce problème où l'on recherche simultanément toutes les occurrences de plusieurs motifs dans un seul texte.

Ce papier présente un algorithme on-line efficace de recherche-dictionnaire lorsque les motifs sont de même longueur et l'alphabet des entrées non borné. Une borne inférieure "précise" montre que notre approche est optimale sur l'ensemble des algorithmes qui procèdent par comparaisons deux à deux.

Une application immédiate est l'utilisation de ce nouvel algorithme de recherche-dictionnaire à un algorithme de recherche multidimensionnelle connu précedemment. Cela améliore les performances de cet algorithme sur les alphabets non bornés. L'algorithme ainsi obtenu est actuellement le plus rapide pour la recherche en tableau k-dimensionnelle sur des alphabets non bornés, pour $k \geq 3$.

*The author was partially supported by the European Research Consortium for Informatics and Mathematics postdoctoral fellowship. Part of this work was done while visiting at the Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands.

1 Introduction

Given a collection of pattern strings $\mathcal{D} = \{\mathcal{P}_1, \mathcal{P}_2, \cdots, \mathcal{P}_{|D|}\}$, one is interested in finding all occurrences of the patterns in a text string $\mathcal{T}$. This problem, which is called the *dictionary-matching* problem or the *multi-pattern string-matching* problem, is a generalization of the standard string-matching problem where only occurrences of a single pattern are sought. For a survey on pattern matching algorithms see Aho's paper [1].

The collection of patterns $\mathcal{D}$ will be called a *dictionary* and its size is denoted by $d = |D|$. The length of the text is $n = |T|$ and the length of each pattern is $|P_i|$. When there is only one pattern or if all patterns in the dictionary have the same length, we use m to denote this length. The number of distinct symbols in the dictionary, which is called the *dictionary alphabet size*, is denoted by $\sigma_{\mathcal{D}}$. Clearly $\sigma_{\mathcal{D}} \leq \Sigma_{i=1}^{d} |P_i|$.

The assumptions on the alphabet that the input symbols are chosen from has a crucial role in the design of efficient string algorithms. To solve the string-matching or the dictionary-matching problems one only has to be able to compare pairs of input symbols in order to get $(=, \neq)$ answers. However, since alphabet symbols are encoded numerically on a computer, the symbols are naturally (arbitrarily) ordered. This order can be used to obtain more efficient algorithms that access the input strings by $(<, =, >)$-comparisons. Furthermore, sometimes it is convenient to assume that the alphabet symbols are small integers which are bounded by some function of the input size, and in most practical cases the alphabet symbols are integers from a constant range (i.e. *ASCII* encoded characters). For a brief comparison of the common assumptions that are made about the input alphabet in the literature see Breslauer's thesis [18].

The string-matching problem has several linear time algorithms. For instance, the classical string-matching algorithm of Knuth, Morris and Pratt [29] takes $O(m)$ time to pre-process the pattern and then $O(n)$ time to find all occurrences of the pattern in the text. The naive approach to the dictionary-matching problem could try to match each pattern separately by using this algorithm, resulting in a dictionary-matching algorithm that has an $O(\Sigma_{i=1}^{d} |P_i|)$ time dictionary preprocessing step and an $O(dn)$ time text scanning step. Thus, the dictionary-matching problem can be solved in $O(\Sigma_{i=1}^{d} |P_i| + dn)$ time.

Aho and Corasick [2] generalized the Knuth-Morris-Pratt string-matching algorithm and showed that on a constant size alphabet, the dictionary-matching problem can be solved by constructing an automaton in $O(\Sigma_{i=1}^{d} |P_i|)$ time and then scanning the text in $O(n)$ time[1] using the automaton. This is a significant improvement since in practice the text can be very long and since the same static pattern dictionary is often searched in several text strings. On unbounded ordered alphabets the Aho-Corasick algorithm takes $O((\Sigma_{i=1}^{d} |P_i| + n) \log \min(d, \sigma_{\mathcal{D}}))$ time[2]. It is interesting to note that the Aho-Corasick algorithm is optimal in the case of a constant size alphabet and almost optimal if the only access to the input strings is by $(=, \neq)$-comparisons[3].

[1]The time bounds for dictionary-matching algorithms usually includes also an additive factor of T_{occ}, the total number of occurrences of the dictionary patterns in the text (clearly $T_{occ} \leq dn$). More precisely, the text scanning step takes $O(n + T_{occ})$ time if all occurrences of the patterns have to be reported. However, in many cases it is sufficient to report some representation of all the occurrences, such as the longest pattern starting or ending at each text position, and thus to save the $O(T_{occ})$ time factor.

[2]In this paper the $\log n$ function usually means $\max(1, \log_2 n)$.

[3]If $n \geq (1 + \epsilon) \max_{i=1..d} |P_i|$, for some constant $\epsilon > 0$, then the text scanning step requires at least $\Omega(nd)$ $(=, \neq)$-comparisons. This bounds is achieved by the Aho-Corasick algorithm and also by the naive algorithm that matches each pattern separately. When the text is shorter, it is possible to modify the text scanning step of the Aho-Corasick

Comments-Walter [22, 23] gave another dictionary-matching algorithm that is based on ideas from the Boyer-Moore [17] string-matching algorithm. This algorithm achieves faster average running times by matching the patterns from their end towards their start. Recently, the dictionary-matching problem gained interest after the discovery of algorithms that can handle dictionaries that are dynamically changing without having to recompute the dictionary preprocessing information from scratch [7, 8, 9, 10, 11, 27].

The main contribution of this paper is a new approach to the dictionary-matching problem on ordered alphabets. Similarly to the Commentz-Walter [22, 23] algorithm, the new algorithm tries to match the dictionary patterns from their end towards their start. However, our motivation is entirely different. While the Commentz-Walter algorithm matches the patterns from their end to start hoping to skip parts of the text, our algorithm does so since this order of comparisons allows to replace a logarithmic multiplicative factor in the running time of the Aho-Corasick algorithm by an additive factor and thus, to amortize some comparisons against large segments of the text.

The new dictionary-matching algorithm takes $O((\frac{\log d}{m}+1)n)$ time for scanning the text after an $O(dm \log \sigma_{\mathcal{D}})$ time dictionary preprocessing step. A tight lower bound shows that the text scanning step is the fastest possible if the algorithm can access the symbols of the input strings only by pairwise $(<,=,>)$-comparisons. The suggested implementation of the algorithm in the standard random-access machine model [3] requires a large sparse table which is used for fast access to the preprocessing data. We suggest two alternatives to store this table. The first uses $O(md^2)$ space that does not need to be initialized, and therefore is time efficient. The second alternative uses a hashing scheme of Fredman, Komlos and Szemeredy [25] and requires only $O(md)$ space, at the cost of introducing randomization in the dictionary preprocessing step. The preprocessing that creates the hash table takes $O(md)$ time with high probability [24]. Note that the rest of the dictionary preprocessing step takes $O(md \log \sigma_{\mathcal{D}})$ time, which dominates the time it takes to create the hash table with very high probability.

In the k-dimensional array-matching problem, the pattern and the text are k-dimensional arrays of sizes m^k and n^k respectively[4]. For a survey on array-matching algorithms see Amir's paper [4]. Bird [16] and Baker [14] demonstrated that the k-dimensional array-matching problem can be solved by k iterations of a one-dimensional dictionary-matching algorithm. By using the Aho-Corasick algorithm they were able to obtain an $O(m^2+n^2)$ time two-dimensional and an $O(m^k+n^k \log m)$ time k-dimensional algorithms on constant size alphabets. When the alphabet is larger, the time bounds become $O(m^k \log m + n^k \log m)$. Other algorithms for the two-dimensional and the k-dimensional problems were suggested by Karp and Rabin [28], Zhu and Takaoka [34] and by Baeza-Yates and Régnier [13].

Amir, Benson and Farach [6] designed a two-dimensional array-matching algorithm that does not resort to one-dimensional techniques. Their algorithm has an $O(n^2)$ time text processing step that uses only $(=,\neq)$-comparisons. However, their pattern processing step still uses one-dimensional techniques and takes $O(m^2 \log m)$ time using $(<,=,>)$-comparisons. Galil and Park [26] improved the preprocessing step by giving an $O(m^2)$ time algorithm that uses only $(=,\neq)$-comparisons.

We show that with the new dictionary-matching algorithm it is possible to implement the Bird-Baker approach with an $O(n^k)$ time text scanning step and an $O(m^k \log m)$ time dictionary preprocessing using $(<,=,>)$-comparisons. This algorithm improves on the best previous bounds

algorithm to be optimal by ignoring patterns that are too long to occur in the text.

[4]The discussion in this paper assumes that the pattern and the text are k-dimensional cubes. It trivially generalizes to any rectangular k-dimensional arrays.

for k-dimensional array-matching on unbounded alphabets, for $k \geq 3$.

The paper is organized as follows. In Section 2 we present the dictionary-matching algorithm on unbounded alphabets. Section 3 shows that the new algorithm can be used in the Bird-Baker higher-dimensional array-matching algorithm. We conclude with list of open problems in Section 4.

2 Dictionary-Matching on Ordered Alphabets

This section first outlines the ideas behind the dictionary-matching algorithm for uniform length dictionaries on unbounded ordered alphabets. The discussion proceeds in the comparisons model and it is simple and elegant. The implementation details of the algorithm in the standard random-access machine model [3] are more involved and given in Section 2.3.

2.1 The new algorithm

Without loss of generality assume that the text is of length $2m-1$. We say that there is a *potential occurrence* of a pattern $\mathcal{P}_i$ starting at text position t if such an occurrence has not been ruled out by the results of previous comparisons. Initially, there is a potential occurrence of each pattern in the dictionary starting at all text positions t, such that $1 \leq t \leq m$. Thus, there are a total of dm potential occurrences. See Figure 1.

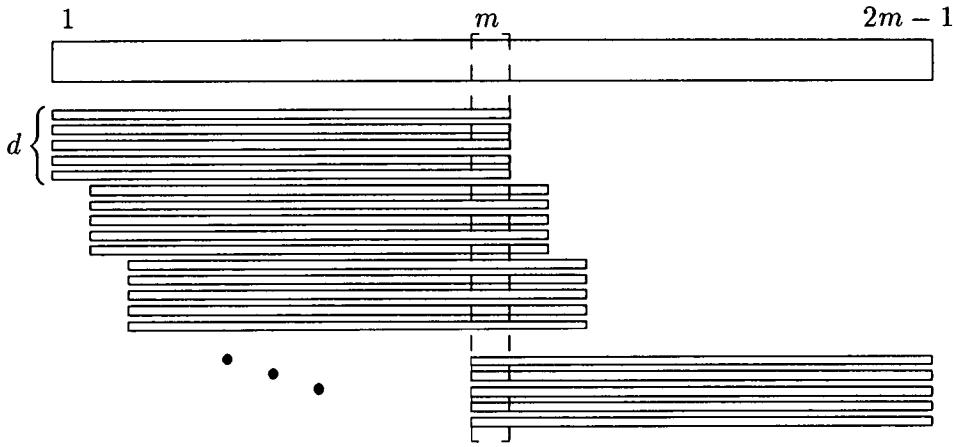


Figure 1: The dm potential occurrences in a text of length $2m-1$.

The Aho-Corasick dictionary-matching algorithm tries to match the patterns from their start towards their end. It might take up to $\log d$ comparisons to determine that none of the patterns in the dictionary occurs starting at the first text position. Even when the Aho-Corasick algorithm reaches this conclusion it has eliminated only d potential occurrences. Up to $m \log d$ comparisons might be required to eliminate all the potential occurrences.

The new dictionary-matching algorithm will start matching the patterns from their end towards their start. The algorithm starts in a *backward phase* from text position m and proceeds towards the beginning of the text. Later, it will continue in a *forward phase* to match the remainder of the text starting from text position $m+1$ and proceeding towards the end of the text. In both

phases, the algorithm advances to the next text position only after it has compared the symbol in the current text position successfully to some pattern symbol.

Consider all potential occurrences aligned with the text starting at the text position where they might occur (Figure 1). The algorithm begins in a backward step by comparing the symbol in the current text position, m, to the median of the alphabet symbols that appear in the potential occurrences in the column which is aligned with this text position.

If the comparison results in an equal answer, the algorithm "knows" which pattern symbol appears in the current text position and it will not compare it again. The algorithm eliminates all potential occurrences which do not agree with the result of the comparison and proceeds to the next text position.

After the algorithm moves to a smaller text position, there might be some potential occurrences which have not been eliminated, but do not intersect the column under the current text position. These are still valid potential occurrences and are considered inactive for the moment. These potential occurrences become active again when the algorithm starts the forward phase.

If the comparison results in an unequal answer, either less-than or greater-than, then the algorithm also eliminates all potential occurrences which do not agree with the result of the comparison. Since the compared dictionary symbol was the median, at least half of the active potential occurrences are eliminated. (Sometimes, by far more than the number of potential occurrences that might be eliminated in one step of the Aho-Corasick algorithm.) The algorithm then continues comparing the symbol at the current text position to the median of the alphabet symbols that appear in the surviving potential occurrences in the column which is aligned with this text position.

The algorithm proceeds this way in the backward phase until the beginning of the text or until all potential occurrences which start at the current text position and at smaller positions, are eliminated. Then, the algorithm starts the forward phase and proceeds similarly. After the algorithm moves to next text position, there might also be some potential occurrences which have not been eliminated, but do not intersect the column under the current text position. Only now, if a potential occurrence does not intersects the column under the current text position, then this potential occurrence is an actual occurrence and it may be reported as such.

The number of potential occurrences at the beginning of the forward phase is obviously not larger than dm, their initial number. Clearly, the algorithm does not make more than $2m - 1$ comparisons that result in an equal answer and no more than $2 \log dm$ comparisons which result in an unequal answer.

Thus, the total number of comparisons is bounded by $O(\log d + m)$ and the $\log d$ multiplicative factor of the Aho-Corasick algorithm was converted to an additive factor that disappears if $\log d = O(m)$.

2.2 Tight bounds

We summarize the ideas presented above and prove the following tight bounds.

Theorem 2.1 *The dictionary matching problem with uniform length dictionaries on ordered alphabets takes* $\Theta((\frac{\log d}{m} + 1)n)$ *comparisons after preprocessing.*

The theorem follows from the next two lemmas.

Lemma 2.2 *The dictionary-matching problem with uniform length dictionaries on ordered alphabets can be solved using* $O((\frac{\log d}{m}+1)n)$ *comparisons. The dictionary preprocessing that is required for the algorithm uses* $O(dm \log \sigma_{\mathcal{D}})$ *comparisons.*

Proof: Partition the text into $O(n/m)$ overlapping blocks of length $2m-1$ (the last block might be shorter) in such a way that each occurrence of a pattern in the text is exactly in one such block. The blocks are processed one by one starting from the beginning of the text. We have shown that the number of comparisons made in each such block is bounded by $O(\log d + m)$ and thus, the number of comparisons required for the whole text is $O((\frac{\log d}{m}+1)n)$.

The dictionary preprocessing has to compute the medians used in the text scanning step. This can obviously be done in the comparison model by sorting all the pattern symbols using $O(dm \log \sigma_{\mathcal{D}})$ comparisons [3]. □

An important property of the Aho-Corasick dictionary-matching algorithm is that it processes the text symbols from left to right and it does not go back to access previous symbols. This means that the text symbols can be processed immediately after they are given as input and there is no need to store them. Note, that an occurrence of a pattern starting at a certain text position can not be reported before the next m text symbols have been processed. We say that a dictionary-matching algorithm for uniform length dictionaries is *on-line* if the algorithm reports whether there is an occurrence of a dictionary pattern starting at text position t before examining symbols in any text positions that are larger than or equal to $t+m$. The algorithm in Lemma 2.2 is clearly on-line.

The lower bound that we prove next only applies to the text scanning step. This lower bound holds for any algorithm that has access to the symbols of the input strings only by pairwise ($<, =, >$)-comparisons. We assumes that pairwise comparisons of text symbols are not permitted.

Lemma 2.3 *The dictionary-matching problem requires at least* $\Omega((\frac{\log d}{m}+1)n)$ *comparisons.*

Proof: Assume that all the symbols of the pattern strings $\mathcal{P}_1, \cdots, \mathcal{P}_d$ are different and are ordered arbitrarily. Partition the text into $O(n/m)$ non overlapping consecutive blocks of length $2m-1$ each (the last block might be shorter). The lower bound is proved for each block separately.

Initially, each pattern $\mathcal{P}_i$ can occur at text positions between positions 1 and m of each block. Thus, there are dm potential occurrences of patterns in a text block. Each comparison between a pattern symbol and a text symbol can be answered less than, equal to, or greater than, in such a way that at least a third of the potential occurrences survive. Therefore, at least $\log_3 dm$ comparisons are required until there is exactly one potential occurrence left.

The algorithm has to make at least m comparisons to verify that there is an actual occurrence. Thus, $\Omega(\max(\log dm, m)) = \Omega(\log d + m)$ comparisons are required in each block of length $2m-1$. The total number of comparisons required in the whole text is $\Omega((\frac{\log d}{m}+1)n)$. □

2.3 Implementation details

This section shows how to implement the comparison model algorithm that was described above in the standard random access machine model [3]. The implementation uses suffix trees in a similar way to Amir and Farach [7] and Amir et al. [9].

2.3.1 Suffix trees

Suffix trees are a form of *digital search trees* that is very useful in many algorithms on strings. The usual definition of a suffix tree is the following:

Definition 2.4 *Let $\mathcal{S}[1..k]$ be a string whose last symbol $\mathcal{S}[k]$ = '#' and '#' is a special alphabet symbol that does not appear anywhere else in $\mathcal{S}$. The* suffix tree $T_{\mathcal{S}}$ *of $\mathcal{S}[1..k]$ is a rooted tree with k leaves and $k-1$ internal nodes such that:*

1. *Each edge is labeled with a non-empty substring of $\mathcal{S}$.*
2. *No two sibling edges are labeled with substrings that start with the same symbol.*
3. *Each leaf is labeled with a distinct position of $\mathcal{S}$.*
4. *The concatenation of the labels of the edges on the path from the root to a leaf labeled with position i is equal to the suffix $\mathcal{S}[i..k]$.*

An example of a suffix tree is given in Figure 2. Note that a suffix tree has no internal nodes with one child. Substrings of $\mathcal{S}[1..k]$ can be represented by their starting and ending positions and therefore, a suffix tree $T_{\mathcal{S}}$ can be stored in $O(k)$ space.

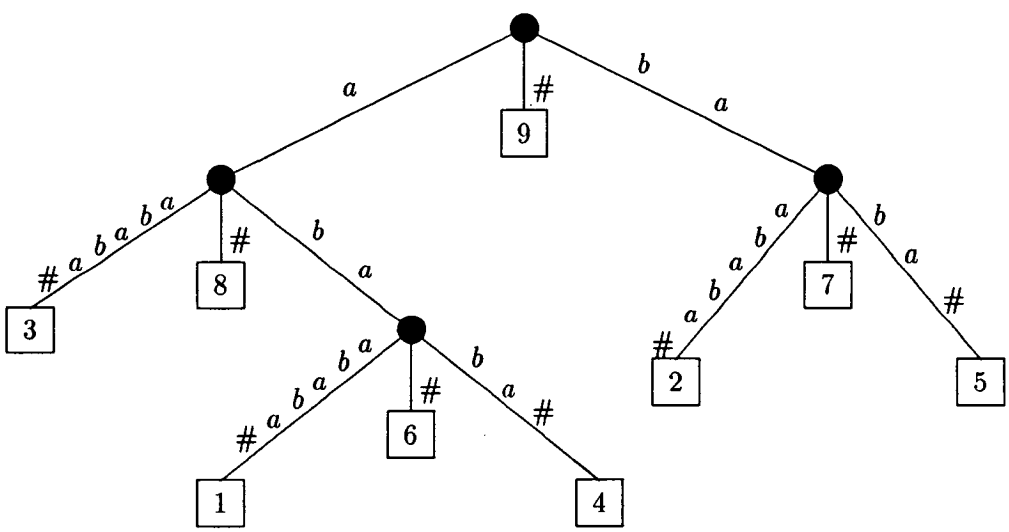


Figure 2: The suffix tree of $\mathcal{S}[1..9]$ = '*abaababa*#'. The edges are labeled with substrings of $\mathcal{S}$ and the leafs with the position in $\mathcal{S}$ at which the corresponding suffix starts.

The important property of suffix trees that we use in this paper is that if two suffixes $\mathcal{S}[i..k]$ and $\mathcal{S}[j..k]$ have the same prefix, namely if $\mathcal{S}[i..i+l-1] = \mathcal{S}[j..j+l-1]$ and $\mathcal{S}[i+l] \neq \mathcal{S}[j+l]$, then the paths from the root to the leafs labeled with positions i and j share an initial segment. The concatenation of the labels of the edges on this initial segment is equal to $\mathcal{S}[i..i+l-1]$. The special alphabet symbol '#' that was assumed to be the last symbol of the string $\mathcal{S}[1..k]$ is normally appended at the end of a given string to guarantee that the suffix tree has distinct leaves that correspond to each suffix. There exist several efficient algorithms to construct suffix trees [19, 30, 33].

Theorem 2.5 *Given a string $S[1..k]$, the suffix tree T_S can be built in $O(k \log \sigma_S)$ time using $(<, =, >)$-comparisons, where σ_S is the number of distinct symbols in S.*

We generalize the notion above to deal with several strings. Given a collection of strings $\{S_1, \cdots, S_p\}$, $S = S_1 \text{`}\#_1\text{'} \cdots S_p \text{`}\#_p\text{'}$ is obtained by concatenating the strings and special distinct alphabet symbols $\text{`}\#_1\text{'}, \ldots, \text{`}\#_p\text{'}$ that do not appear anywhere in the strings. The suffix tree T_S identifies identical prefixes of suffixes of these strings and has distinct leafs that correspond to each suffix of $S_i \text{`}\#_i\text{'}$, for $i = 1, \cdots, p$. Note that the concatenation of the labels of edges from the root to a leaf is equal to a suffix of one of the strings S_i followed by the complete strings S_j, $j = i+1, \cdots, p$, and the special alphabet symbols that are appended at the end of these strings. The special alphabet symbols $\text{`}\#_i\text{'}$ are identified by their position within S and there is no need represent them as real alphabet symbols since they are not equal to any other symbol.

Lemma 2.6 *Given a collection of strings $\{S_1, \cdots, S_p\}$, we construct a data structure that is used to find all prefixes of a string Q that are equal to suffixes of some of the given strings.*

Let σ_S denote the number of distinct symbols in the strings $\{S_1, \cdots, S_p\}$ and let $l = \Sigma_{i=1}^{p} |S_i|$ denote the sum of their lengths. Then, the data structure is constructed in $O(l \log \sigma_S)$ time and uses $O(l)$ space. The prefixes of $Q[1..q]$ are reported on-line in $O(q + \log l)$ time using $(<, =, >)$-comparisons.

Proof: Given a collection of strings $\{S_1, \cdots, S_p\}$, we construct the suffix tree T_S for the string $S = S_1 \text{`}\#_1\text{'} \cdots S_p \text{`}\#_p\text{'}$, where $\text{`}\#_1\text{'}, \ldots, \text{`}\#_p\text{'}$ are special distinct alphabet symbols that do not appear anywhere in the given strings and in $Q[1..q]$. This suffix tree together with some auxiliary data structures that we describe next is used to find all prefixes of $Q[1..q]$ that are equal to suffixes of some of the given strings.

If several suffixes of strings $S_1, \cdots, S_p$ are equal, then the leafs in T_S that are associated with these suffixes have the same parent node. Using this property, it is possible to identify all suffixes of the strings $S_1, \cdots, S_p$ that are equal and assign unique names to equal suffixes. We agree that the unique name for a suffix is defined as the 2-tuple that contains the suffix length and the smallest index of a string in the collection that has the same suffix. It is possible to find all the equal suffixes and assign the unique names to the suffixes by a simple traversal of the suffix tree T_S in $O(l)$ time.

Since we can not hope to report all strings that have a given suffix within the required time bounds, we will report only the unique name that was assigned to each suffix that is found. Using this unique name, it is possible to find the rest of the strings that have the same suffix, if necessary.

The algorithm proceeds by traversing the suffix tree T_S starting from the root, while Q is given on-line a symbol at a time. The algorithm follows an edge from a node v to a node w in the suffix tree if the label on this edge is equal to the next symbols of Q. Since sibling edges in the suffix tree are labeled with substrings that start with different symbol, the algorithm can decide which edge to follow based only on the current symbol of Q. The algorithm then continues and checks that the complete label of that edge is equal to the next symbols of Q.

When the traversal reaches a node v of the suffix tree, the algorithm has to report if it found a suffix of any of the strings $S_1, \cdots, S_p$. If there is an an edge from v to a leaf, and the edge has a label that starts with one of the special symbols $\text{`}\#_i\text{'}$, then a suffix of the string S_i was found. Note that there might be many strings that have this same suffix, and as agreed before, we report only the smallest index of a string that has this suffix (this index was precomputed and associated with the node v of the suffix tree).

It is also possible that the algorithm did not find an edge whose label matches the next symbols of $\mathcal{Q}$. In this case the algorithm terminates since it is not possible that longer prefixes of $\mathcal{Q}$ are equal to suffixes of the strings $\mathcal{S}_1, \cdots, \mathcal{S}_p$. However, the algorithm might have found an edge whose label has a nonempty prefix that matches the next symbols of $\mathcal{Q}$. If the label on this edge matches the next symbols of $\mathcal{Q}$ up to a special symbol '$\#_i$', then the algorithm reports also that this prefix of $\mathcal{Q}$ is equal to a suffix of $\mathcal{S}_i$. Note that in this case there is only one suffix to be reported.

When the algorithm has to choose which edge to follow from a node, it can decide based only on the first symbols of the edge labels. Normally, if $(<, =, >)$-comparisons are used, one can search for the current symbol of $\mathcal{Q}$ in the list of first symbols of the edge labels using binary search or binary search trees [3, 32]. This binary search is the cause of the $\log \sigma_{\mathcal{S}}$ multiplicative factor in the running time of the suffix tree construction algorithm and also in the running time of the Aho-Corasick dictionary-matching algorithm. Note, that since the special symbols '$\#_i$' do not appear in $\mathcal{Q}$, they do not have to be considered in this binary search.

However, a binary search does not give us the bounds we are trying to achieve. We rather proceed in the spirit of the algorithm given in Section 2.1 and create a data structure that will be used to guide the search for the edge that the algorithm follows, more efficiently.

Given a suffix tree $T_{\mathcal{S}}$, we define the weight of a node v as the number of leafs in the subtree rooted at v. The weights of all nodes can be computed by a simple traversal of the suffix tree in $O(l)$ time.

In order to find the edge whose label starts with the current symbol of $\mathcal{Q}$, we use a weighted search. Similarly, to the binary search, the edges at a node v are ordered by the first symbols on their labels. However, instead of choosing the median of the remaining possible edges, we choose the weighted median where each edge has the weight of the node it is leading to. This way, instead of eliminating half of the possible edges in case of a mismatch, we eliminate half of the weight. Since the weight of the current node that the algorithm considers only decreases when the algorithm find the right edge and follows it to the next node, we are guaranteed that there are no more than $\log l$ mismatches throughout the algorithm. Since there are up to q comparisons that result in equal answers and all other steps take constant time, the total time spent is $O(q + \log l)$.

To implement the weighted search efficiently, we precompute a search tree for each node of the suffix tree $T_{\mathcal{S}}$. The search trees in all the nodes are computed in $O(l \log \sigma_{\mathcal{S}})$ time and stored in $O(l)$ space. □

The same data structure can also be used to find all suffixes of a string $\mathcal{Q}[1..q]$ that are equal to prefixes of some strings $\{\mathcal{S}_1, \cdots, \mathcal{S}_p\}$ by considering the reversed strings.

2.3.2 The implementation

Recall that the dictionary patterns have a uniform length m, and assume without loss of generality that the text $\mathcal{T}$ is of length $2m - 1$. The backward and forward phases are implemented independently and symmetrically. Then, their results are combined to report the occurrences of the patterns in the text.

The dictionary preprocessing builds the data structure described in Lemma 2.6 to find all prefixes of the dictionary patterns that occur at the end of $\mathcal{T}[1..m]$ and all suffixes of the dictionary patterns that occur at the beginning of $\mathcal{T}[m + 1..2m - 1]$. The occurrences of patterns in the text are reported by checking if the concatenation of some of the prefixes that were found with some of the suffixes is equal to some patterns in the dictionary.

To check efficiently if the string obtained by the concatenation of the pattern prefix $\mathcal{P}_k[1..i]$ with the pattern suffix $\mathcal{P}_l[i+1..m]$ is equal to some dictionary pattern, the dictionary preprocessing step computes a table $presuf_i(k,l)$ the gives the index of a dictionary pattern that is equal to prefix-suffix concatenation if such a pattern exists. If there are several patterns in the dictionary that are equal, we agree that we report only a representative which is the smallest index of a pattern that was found. If there is a need to report the indices of all patterns an additional list data structure is created to identify all equal patterns.

The $presuf_i(k,l)$ table is defined for $i = 1,\cdots,m$ and $k,l = 1,\cdots,d$ and it seems to require $O(md^2)$ space. However, since the data structure in Lemma 2.6 reports a unique representative for each pattern prefix or suffix, and since there are only d strings in the dictionary, the table $presuf_i(\cdot,\cdot)$, has only d relevant entries, for each $i = 1,\cdots,m$. Therefore, the whole table has $O(md)$ relevant entries. Using the hashing scheme of Fredman, Komlos and Szemeredy [25] it is possible to represent this table in $O(md)$ space with a constant time access, after an initialization step that is done in the dictionary preprocessing step and takes $O(md)$ time with high probability [24]. Alternatively, given a constant $\epsilon > 0$, this table can be represented in $O(md^{1+\epsilon})$ space with an $O(1/\epsilon)$ access time similarly to Apostolico et al. [12].

Lemma 2.7 *There exists a dictionary-matching algorithm for uniform length dictionaries with d patterns of length m and a text of length $2m-1$ that takes $O(\log d + m)$ time using $(<,=,>)$-comparisons. The dictionary preprocessing takes $O(md\log\sigma_{\mathcal{D}})$ time and uses $O(md^2)$ space.*

Proof: The pattern preprocessing first creates the data structure that was described in Lemma 2.6 for the forward and backward phases. These data structures are created in $O(md\log\sigma_{\mathcal{D}})$ time and use $O(md)$ space. The $presuf_i(k,l)$ table is created in $O(md)$ time and uses $O(md^2)$ space.

The text scanning step uses these data structures to find the prefixes and suffixes of dictionary patterns in the text in $O(\log d+m)$ time. It then reports the occurrences of the dictionary patterns using the $presuf_i(k,l)$ table in $O(m)$ time. □

The lemma above is used repeatedly when the text is longer. The following theorem summarizes the properties of the new dictionary matching algorithm.

Theorem 2.8 *There exists an on-line dictionary-matching algorithm for uniform length dictionaries with d patterns of length m and a text of length n that takes $O((\frac{\log d}{m}+1)n)$ time using $(<,=,>)$-comparisons.*

3 Higher Dimensional Array Matching

Bird [16] and Baker [14] independently discovered a higher-dimensional array-matching algorithm that uses a one-dimensional dictionary-matching algorithm as a subroutine. Their original algorithm used the Aho-Corasick [2] algorithm that was available at the time, but any dictionary-matching algorithm can be used. It particular, by using the dictionary-matching algorithm that was described in Section 2.3 it is possible to improve the running time of the Bird-Baker array-matching algorithm on unbounded ordered alphabets.

Bird and Baker reduced the k-dimensional array matching problem into k iterations of a one-dimensional dictionary matching problems. We explain how their algorithms works in two-dimensions.

Assume that the pattern is given as $\mathcal{P}[1..m, 1..m]$ and the text is $\mathcal{T}[1..n, 1..n]$. Note that each row or column of a two-dimensional array can be regarded as a one-dimensional string.

1. Denote the pattern rows $\mathcal{P}_i = \mathcal{P}[i, 1..m]$. The algorithm starts by solving a dictionary matching problem that finds all occurrences of the uniform length pattern rows $\mathcal{P}_i$, $i = 1..m$, in each row of the text.

   In fact, since some of the pattern rows might be identical, the algorithm first finds a unique representative for each set of equal pattern rows and searches only for this representative. Namely, $\mathcal{P}_{u(i)}$ is the representative for pattern row $\mathcal{P}_i$, such that $\mathcal{P}_{u(i)} = \mathcal{P}_i$ and $u(i) = u(j)$, for all pattern rows $\mathcal{P}_i = \mathcal{P}_j$. The pattern is obviously represented as:

$$\mathcal{P}[1..m, 1..m] = \begin{array}{l} \mathcal{P}_{u(1)} \\ \mathcal{P}_{u(2)} \\ \cdot \\ \cdot \\ \mathcal{P}_{u(m)}. \end{array}$$

   The output of the dictionary matching problem is an array $\mathcal{T}_1[1..n, 1..n]$ such that $\mathcal{T}_1[j, k] = u(i)$ if an occurrence of $\mathcal{P}_{u(i)}$ was discovered in the text row starting at $\mathcal{T}[j, k]$, and $\mathcal{T}_1[j, k] = 0$ if no occurrence of a pattern row starts at $\mathcal{T}[j, k]$. More formally, $\mathcal{T}_1[j, k] = u(i)$ if and only if $\mathcal{T}[j, k + l - 1] = \mathcal{P}[u(i), l]$, for $l = 1..m$.

   The unique representatives $\mathcal{P}_{u(i)}$ can be found while the dictionary of the pattern rows $\mathcal{P}_i$, $i = 1..m$, is created. The dictionary preprocessing takes $O(m^2 \log m)$ time and the text scanning step takes $O(n^2)$ time.

2. Clearly, there is an occurrence of the pattern $\mathcal{P}[1..m, 1..m]$ starting at $\mathcal{T}[j, k]$ if and only if $\mathcal{T}_1[j + l - 1, k] = u(l)$, for $l = 1..m$.

   Therefore, the occurrences of the pattern can be discovered by using a string matching algorithm to search for occurrences of the sequence $u(1)u(2)\cdots u(m)$ in each column of $\mathcal{T}_1[1..n, 1..n]$.

   These occurrences can be found by using the Knuth-Morris-Pratt string-matching algorithm with $O(m)$ time pattern preprocessing and $O(n^2)$ time text scanning.

Thus, the overall time complexity of the Bird-Baker algorithm using the new dictionary-matching algorithm is $O(n^2 + m^2 \log m)$. The extension to k-dimensions is straightforward. It requires $O(n^k)$ time for the text scanning step and $O(m^k \log m)$ time for the pattern preprocessing. It is important to note the following two facts:

1. The last iteration of the Bird-Baker algorithm searches only for a single pattern and can use the Knuth-Morris-Pratt string matching algorithm.

2. The matching problems solved after the first iteration of the Bird-Baker algorithm do not involve the input alphabet. Therefore, even if the input alphabet is of constant size, the alphabet size in the later iterations might be larger.

3.1 Remarks

1. In the discussion above we assume that the input arrays are of a fixed dimension k. This allows the *Big-O* notation to hide some multiplicative factors that depend of k.

   In fact, all published higher-dimensional array-matching algorithms that we are familiar with, including the Bird-Baker algorithm, require in one form or another k iterations over the text array. Therefore, the time complexity bounds of these algorithms include a multiplicative factor of k. A second multiplicative factor is usually hidden in the fact that indexing a k-dimensional array required $O(k)$ time. However, in many cases it is possible to avoid this dependence on k by paying a careful attention to the access pattern and by using the one-dimensional memory representation of arrays.

   The bounds of our algorithm sometimes hide a few more multiplicative k factors.

2. The discussion above clearly generalizes to rectangular k-dimensional array. However, to obtain good time bounds one has to pay attention to the direction, or order of coordinates, in which the arrays are processed.

   For example, suppose that the pattern is an $m_1 \times m_2$ two-dimensional array where $m_1 \ll m_2$ and the text is an $n \times n$ square. The Bird-Baker approach solves a dictionary-matching problem followed by a string matching problem. The time complexity of the dictionary-matching problem depends on the direction in which the algorithm proceeds: $O((\frac{\log m_1}{m_2}+1)n^2)$ time for m_1 pattern strings of length m_2 each, or alternatively, $O((\frac{\log m_2}{m_1}+1)n^2)$ time for m_2 pattern strings of length m_1 each. Clearly, the first direction is better, yielding an $O(n^2)$ algorithm.

   The same principle holds in higher dimensions. If the coordinates are ordered in such a way that the pattern is an $m_1 \times \cdots \times m_k$ array with $m_1 \leq \cdots \leq m_k$ and the text is an $n_1 \times \cdots \times n_k$ array, then the text scanning step takes $O(n_1 \times \cdots \times n_k)$ time and the pattern preprocessing step takes $O(m_1 \times \cdots \times m_k \times \log m_k)$ time.

3. It is interesting to point out that there exist a trivial comparison model algorithm for k-dimensional array-matching using $2n^k + 2m^k$ $(=, \neq)$-comparisons, including preprocessing. In the standard model, linear bounds are currently achieved only in one and two-dimensions. The complicated details of the two-dimensional algorithm of Galil and Park [26] might suggest that better understanding of periodicity (overlap) properties of higher dimensional arrays [5, 15, 31] would be required before linear time higher dimensional array-matching algorithms are available in the standard model.

4 Open Problems

This work leaves several open questions. Apart from the obvious possibilities for improving the dictionary matching algorithm, such as using only $O(dm)$ space without randomization (hashing), cheaper preprocessing and generalization to variable length dictionaries, the following related problems seem to be of particular interest:

1. Can the ideas that were introduced in this paper help to obtain an efficient dynamic-dictionary-matching algorithm on unbounded alphabets?

2. What is the exact number of comparisons required for the dictionary-matching problem? The number of $(=, \neq)$-comparisons required for the string-matching problem is known almost exactly [20, 21].

3. Is it possible to implement the comparison-model higher-dimensional array-matching algorithm that was mentioned in Section 3.1, in the standard model in linear time?

5 Acknowledgments

I would like to thank Alberto Apostolico, Wojciech Szpankowski and Laura Toniolo for several discussions and comments on this work. I also thank Yossi Matias for his advise on the hashing literature, Zvi Galil for his help in obtaining copies of some bibliography items and Mireille Régnier for the French translation of the abstract.

References

[1] A.V. Aho. Algorithms for Finding Patterns in Strings. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 257–300. Elsevier Science Publishers B. V., Amsterdam, the Netherlands, 1990.

[2] A.V. Aho and M.J. Corasick. Efficient string matching: An aid to bibliographic search. *Comm. of the ACM*, 18(6):333–340, 1975.

[3] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA., 1974.

[4] A. Amir. Multidimensional Pattern Matching (A Survey of Serial Deterministic Algorithms). Manuscript, 1991.

[5] A. Amir and G. Benson. Two-dimensional periodicity and its applications. In *Proc. 3rd ACM-SIAM Symp. on Discrete Algorithms*, pages 440–452, 1992.

[6] A. Amir, G. Benson, and M. Farach. Alphabet-Independent Two-Dimensional Matching. In *Proc. 24th ACM Symp. on Theory of Computing*, pages 59–68, 1992.

[7] A. Amir and M. Farach. Adaptive dictionary matching. In *Proc. 32th IEEE Symp. on Foundations of Computer Science*, pages 760–766, 1991.

[8] A. Amir and M. Farach. Two-dimensional dictionary matching. *Inform. Process. Lett.*, 44:223–239, 1992.

[9] A. Amir, M. Farach, Z. Galil, R. Giancarlo, and K. Park. Dynamic Dictionary Matching. Manuscript, 1992.

[10] A. Amir, M. Farach, R.M. Idury, J.A. La Poutré, and A.A. Schäffer. Improved Dynamic Dictionary-Matching. In *Proc. 4nd ACM-SIAM Symp. on Discrete Algorithms*, pages 392–401, 1993.

[11] A. Amir, M. Farach, and Y. Matias. Efficient Randomized Dictionary-Matching Algorithms. In *Proc. 3rd Symp. on Combinatorial Pattern Matching*, Lecture Notes in Computer Science, pages 262–275. Springer-Verlag, Berlin, Germany, 1992.

[12] A. Apostolico, C. Iliopoulos, G. M. Landau, B. Schieber, and U. Vishkin. Parallel construction of a suffix tree with applications. *Algorithmica*, 3:347–365, 1988.

[13] R. Baeze-Yates and M. Régnier. Fast two-dimensional pattern matching. *Inform. Process. Lett.*, 45:51–57, 1993.

[14] T.P. Baker. A Technique for Extending Rapid Exact-Match String Matching to Arrays of More than One Dimension. *SIAM J. Comput.*, 7(4):533–541, 1978.

[15] G.E. Benson. *Two-Dimensional Periodicity and Matching Algorithms.* PhD thesis, Dept. of Computer Science, University of Maryland, 1992.

[16] R.S. Bird. Two Dimensional Pattern Matching. *Inform. Process. Lett.*, 6(5):168–170, 1977.

[17] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Comm. of the ACM*, 20:762–772, 1977.

[18] D. Breslauer. *Efficient String Algorithmics.* PhD thesis, Dept. of Computer Science, Columbia University, New York, NY, 1992.

[19] M.T. Chen and J. Seiferas. Efficient and elegant subword-tree construction. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume 12 of *NATO ASI Series F*, pages 97–107. Springer-Verlag, Berlin, Germany, 1984.

[20] R. Cole and R. Hariharan. Tighter Bounds on The Exact Complexity of String Matching. In *Proc. 33rd IEEE Symp. on Foundations of Computer Science*, pages 600–609, 1992.

[21] R. Cole, R. Hariharan, M.S. Paterson, and U. Zwick. Which patterns are hard to find. In *Proc. 2nd Israeli Symp. on Theoretical Computer Science*, pages 59–68, 1993.

[22] B. Commentz-Walter. A string matching algorithm fast on the average. In H. A. Maurer, editor, *Proc. 6th International Colloquium on Automata, Languages, and Programming*, pages 118–132. Springer-Verlag, Berlin, Germany, 1979.

[23] B. Commentz-Walter. A string matching algorithm fast on the average. Technical Report 79.09.007, IBM Wissenschaftliches Zentrum, Heidelberg, Germany, 1979.

[24] M. Dietzfelbinger, J. Gil, Y. Matias, and N. Pippenger. Polynomial Hash Functions Are Reliable. In *Proc. 19th International Colloquium on Automata, Languages, and Programming.* Springer-Verlag, Berlin, Germany, 1992.

[25] M.L. Fredman, J. Komlos, and E. Szemeredi. Storing a sparse table with O(1) worst case access time. *J. Assoc. Comput. Mach.*, 31(3):538–544, 1984.

[26] Z. Galil and K. Park. Truly Alphabet-Independent Two-Dimensional Pattern Matching. In *Proc. 33th IEEE Symp. on Foundations of Computer Science*, pages 247–256, 1992.

[27] R.M. Idury and A.A. Schäffer. Dynamic Dictionary-Matching with Failure Functions. In *Proc. 3rd Symp. on Combinatorial Pattern Matching*, Lecture Notes in Computer Science, pages 276–287. Springer-Verlag, Berlin, Germany, 1992.

[28] R.M. Karp and M.O. Rabin. Efficient randomized pattern matching algorithms. *IBM J. Res. Develop.*, 31(2):249–260, 1987.

[29] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6:322–350, 1977.

[30] E.M. McCreight. A space economical suffix tree construction algorithm. *J. Assoc. Comput. Mach.*, 23:262–272, 1976.

[31] M. Régnier and L. Rostami. A Unifying Look at d-dimensional Periodicities and Space Coverings. In *Proc. 4rd Symp. on Combinatorial Pattern Matching*, 1993. To appear.

[32] R.E. Tarjan. *Data Structures and Network Algorithms.* SIAM, Philadelphia, PA., 1985.

[33] P. Weiner. Linear pattern matching algorithms. In *Proc. 14th Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[34] R.F. Zhu and T. Takaoka. A Technique for Two-Dimensional Pattern Matching. *Comm. of the ACM*, 32(9):1110–1120, 1989.

Unité de Recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Unité de Recherche INRIA Lorraine Technopôle de Nancy-Brabois - Campus Scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers Lès Nancy Cedex (France)
Unité de Recherche INRIA Rennes IRISA, Campus Universitaire de Beaulieu 35042 Rennes Cedex (France)
Unité de Recherche INRIA Rhône-Alpes 46, avenue Félix Viallet - 38031 Grenoble Cedex (France)
Unité de Recherche INRIA Sophia Antipolis 2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex

ISSN 0249 - 0803